Processor Behavioral Model Simulation

Simulation of Da Vinci v1.0 System Using Verilog

Brandon Feist
CS 147
San Jose State University
San Jose, California

***Abstract*— this paper goes over the Da Vinci system, a simple simulation a processor using the 'CS147DV' Instruction Set and a 32x64M memory module. This paper goes over the separate components and how they function, and then ultimately the Da Vinci v1.0 system as a whole.**

I. Introduction

The goal of the Da Vinci simulation is to create a simple representation as to how a processor and memory interact using Verilog code. The processor consists of an ALU, a 32x32 Register, and a Control Unit. This processor uses the 'CS147DV' Instruction Set to process the given information appropriately. The goal of this project is to go over the ALU, Register, Control Unit, and Memory modules separately to develop an understanding of how they each work, and then ultimately test them together under a single system 'Da Vinci.'

II. System Requirements

This section overviews the system requirements for the Da Vinci V1.0 system. Later sections will go more in depth for each module.

A. ALU

The ALU does the majority of the logic work for the Da Vinci System. It has nine operations that are called using the function code described in the 'CS147DC' Instruction set:

1. Addition
2. Subtraction
3. Multiplication
4. Shift Right
5. Shift Left
6. Logical AND
7. Logical OR
8. Logical NOR
9. Set Less Than

Along with these given operations, the ALU will provide a ZERO flag for the system.

B. Register File

The Register File consist of 32 registers that can store 32 bit word. Reset operations are done on the negative edge of the system clock while the rest of the operations are done on the positive clock edge. DATA_R* is set to X when the READ and WRITE are either 00 or 11.

C. Memory

The Memory Module is consists of 64M of memory storing a 32 bit word at each address. Reset operations are done at the negative edge of the system clock while the rest of the operations are done on the positive edge. DATA is set to X when READ and WRITE are either 00 or 11

D. Control Unit

The control unit works off a 'five-state' state machine that changes between states at every positive clock edge. The control unit goes between states FETCH, DECODE, EXECUTE, MEMORY, and WRITE BACK. The control unit will make decisions according to the 'CS147DV' Instruction set that is applied to the given 32 bit instructions.

E. CS147DV Instruction Set

The CS147DV Instruction Set provides R-Type, I-Type, and J-Type instructions. R-Type instructions consist of three registers in which the target operation is done on two of the registers and then results are place in the third register. I-Type instructions consist of two registers and a 16-bt immediate, where the target operation is done using a register and the immediate, the results are then placed in the second register. J-Type instructions are responsible for jump and stack operations using a given 26bit address.

III. ALU Design and Testing

The system ALU is programmed to provide nine operations and a ZERO flag As stated earlier the nine operations consist of Addition, Subtraction, Multiplication, Shift Right, Shift Left, Logical AND, Logical OR, Logical NOR, and Set Less Than. The needed operations are determined by a given 6-bit operation code, and the operation is applied to a given OP1 and OP2:

A. *ALU Design*

Operations are determined using a case switch and using the given operation code (OPRN) to determine the correct operation.

```
always @(OP1 or OP2 or OPRN)
begin
    case (OPRN)
        `ALU_OPRN_WIDTH'h20 : OUT = OP1 + OP2; // addition
        `ALU_OPRN_WIDTH'h22 : OUT = OP1 - OP2; // subtraction
        `ALU_OPRN_WIDTH'h2c : OUT = OP1 * OP2; // multiplaction
        `ALU_OPRN_WIDTH'h02 : OUT = OP1 >> OP2; // shift right
        `ALU_OPRN_WIDTH'h01 : OUT = OP1 << OP2; // shift left
        `ALU_OPRN_WIDTH'h24 : OUT = OP1 & OP2; // and
        `ALU_OPRN_WIDTH'h25 : OUT = OP1 | OP2; // or
        `ALU_OPRN_WIDTH'h27 : OUT = ~(OP1 | OP2); // nor
        `ALU_OPRN_WIDTH'h2a : OUT = OP1 < OP2 ? 1 : 0; // set less than
        default: OUT = `DATA_WIDTH'hxxxxxxxx;

    endcase
end
```

Upon output (OUT), the results are checked to determine the ZERO flag using a ternary operation.

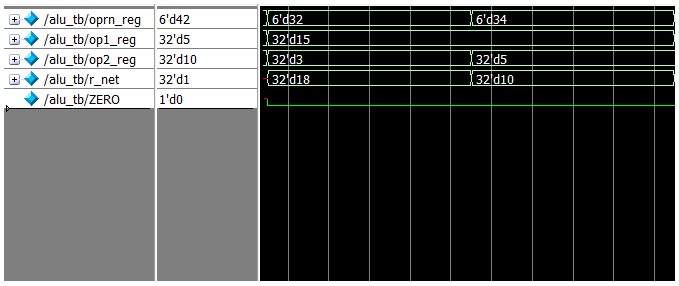
```
always @(OUT)
begin
    ZERO = OUT == 0 ? 1 : 0;
end

endmodule
```

B. *ALU Equation*

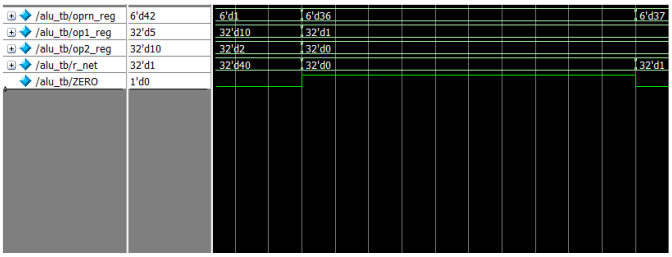
- Addition: OP1 + OP2;
- Subtraction: OP1 – OP2;
- Multiplication: OP1 * OP2;
- Shift Right: OP1 >> OP2;
- Shift Left: OP1 << OP2;
- Logical AND: OP1 & OP2;
- Logical OR: OP1 | OP2;
- Logical NOR: ~ (OP1 | OP2)
- Set Less Then: OP1 < OP2 ? 1 : 0;

C. *ALU Testing*

Testing was done on the ALU using an ALU Test Bench (ALU_TB) to determine if the correct values were placed in the proper OP1 and OP2 inputs and if the correct operations were being applied depending on the given OPRN code.



The ZERO flag was also tracked to determine if it was being set to 1 when the result was 0. As shown in the given wave table, the ZERO flag was working correctly as it was set to 1 during the test for the Logical AND operation (1 & 0) which outputs a result (r_net) of 0.



After the simulation was finished a text output was given from the test bench. Showing that all nine ALU operations passed.

```
# [TEST] 15 + 3 = 18 , got 18 ... [PASSED]
# [TEST] 15 - 5 = 10 , got 10 ... [PASSED]
# [TEST] 5 * 5 = 25 , got 25 ... [PASSED]
# [TEST] 10 >> 2 = 2 , got 2 ... [PASSED]
# [TEST] 10 << 2 = 40 , got 40 ... [PASSED]
# [TEST] 1 & 0 = 0 , got 0 ... [PASSED]
# [TEST] 1 | 0 = 1 , got 1 ... [PASSED]
# [TEST] 21474836 ~| 0 = 4273492459 , got 4273492459 ... [PASSED]
# [TEST] 5 < 10 = 1 , got 1 ... [PASSED]
#
#       Total number of tests           9
#       Total number of pass            9
#
# ** Note: $stop    : C:/Users/Greg/Desktop/prj_02_source/alu_tb.v(113)
#    Time: 95 ns  Iteration: 0  Instance: /alu_tb
```

IV. Register Design And Testing

A. *Register Design*

The register consists of two data output ports (Data_R1 and Data_R2) which can be outputted simultaneously from the specified register location using (ADDR_R1 and ADDR_R2) inputs. There is a data input port (Data_W) in which can be written to a specified register location using (ADDR_W) input. The register determines whether to read or write using the READ and WRITE inputs and will set all DATA_R* to X if both read and write are 0 or 1.

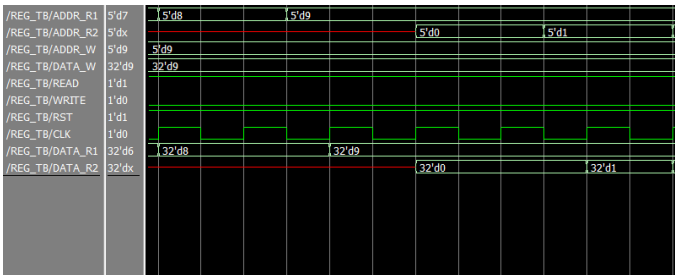
```
//Need to make register do nothing on 00 or 11
assign DATA_R1 = ((READ===1'b1)&&(WRITE===1'b0))?data_ret_1:{`DATA_WIDTH{1'bz} };
assign DATA_R2 = ((READ===1'b1)&&(WRITE===1'b0))?data_ret_2:{`DATA_WIDTH{1'bz} };
```

Reset is programmed to occur on the negative edge of RST, while the rest of the operations occur during the positive edge. The register will read out data when READ=1'b1 and WRITE=1'b0, while on the other hand it will read in data when READ=1'b0 and WRITE=1'b1.

```
always @ (negedge RST or posedge CLK)
begin
  if (RST == 1'b0)
  begin
  for(i=0;i<=`DATA_INDEX_LIMIT; i = i + 1)
      reg_32x32[i] = { `DATA_WIDTH{1'b0} };
end
  else
begin
  if ((READ===1'b1)&&(WRITE===1'b0)) // read op
  begin
        data_ret_1 = reg_32x32[ADDR_R1];
        data_ret_2 = reg_32x32[ADDR_R2];
  end
  else if ((READ===1'b0)&&(WRITE===1'b1)) // write op
        reg_32x32[ADDR_W] = DATA_W;
end
end
```

B. *Register Testing*

When it comes to testing the register, it must be determined if the write and read functions work correctly and are being stored in the correct register addresses. Using the wave table it can be determined of the register is responding correctly according to the system clock and READ and WRITE queues.



The register test bench test the WRITE function by writing (DATA_W) 0-9 to register addresses (ADDR_W) 0-9. Before testing the data read, the register data output is tested while READ and WRITE are set to 0. After the test bench then reads back register addresses 0-9 from the (DATA_R1) output and then the (DATA_R2) output.

```
// test of write data with R1 data
for(i=0;i<10; i = i + 1)
begin
#5      READ=1'b1; WRITE=1'b0; ADDR_R1 = i;
#10      no_of_test = no_of_test + 1;
        if (DATA_R1 !== i)
            $write("[TEST] Read %1b, Write %1b, expecting %8h, got %8h [FAILED]\n", READ, WRITE, i, DATA_R1);
        else
            no_of_pass  = no_of_pass + 1;

end

// test of write data with R2 data
for(i=0;i<10; i = i + 1)
begin
#5      READ=1'b1; WRITE=1'b0; ADDR_R2 = i;
#10      no_of_test = no_of_test + 1;
        if (DATA_R2 !== i)
            $write("[TEST] Read %1b, Write %1b, expecting %8h, got %8h [FAILED]\n", READ, WRITE, i, DATA_R1);
        else
            no_of_pass  = no_of_pass + 1;

end
```

According to the system output from the register test bench (REG_TB), the register passed all 21 data write/read tests determining that it is working correctly.

```
#
#         Total number of tests          21
#         Total number of pass           21
#
```

V. Memory Design And Testing

A. *Memory Design*

Unlike the Register, the Memory unit only provides one inout data port (DATA) in which data can be inputted and outputted from the memory unit. To read from memory READ is set to 1 and WRITE to 0, the data address (ADDR) is given and outputted to the DATA inout port. While to write to memory READ is set to 0 and WRITE to 1, the data address and the data to store is inputted into the DATA inout. The memory is designed to set the DATA to X if both READ and WRITE is 0 or 1.

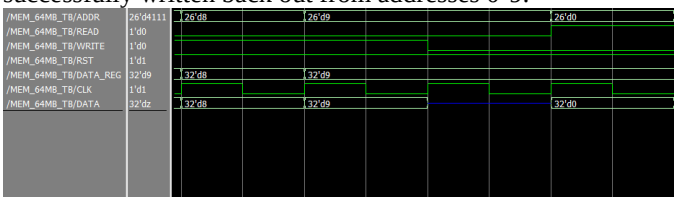
```
begin
  if ((READ===1'b1)&&(WRITE===1'b0)) // read operation
        data_ret =  sram_32x64m[ADDR];
  else if ((READ===1'b0)&&(WRITE===1'b1)) // write operation
        sram_32x64m[ADDR] = DATA;
end
```

Reset is programmed to occur on the negative edge of RST, when RST is set to zero (1'b0). During a reset all of the memory data is set to 0 and then an initialization file (mem_init_file) is read in. The rest of the memory processes are set to occur on the positive edge of the system clock.

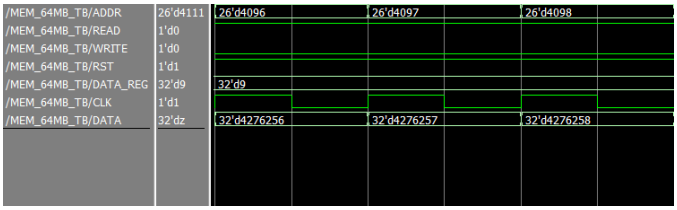
```
assign DATA = ((READ===1'b1)&&(WRITE===1'b0))?data_ret:{`DATA_WIDTH{1'bz} };

always @ (negedge RST or posedge CLK)
begin
if (RST === 1'b0)
begin
for(i=0;i<=`MEM_INDEX_LIMIT; i = i +1)
    sram_32x64m[i] = { `DATA_WIDTH{1'b0} };
$readmemh(mem_init_file, sram_32x64m);
end
```

B. *Memory Testing*

The goal of testing the memory module is to make sure that files are written and read correctly to and from the correct memory address. Using the wave table, after testing the reset operation, it can be seen that data 0-9 is written into memory addresses 0-9 at the positive clock edge. Then the same data is successfully written back out from addresses 0-9.



After checking for written data the test bench checks for loaded memory initialization data which also shows correctly on the wave table.



After finishing the simulation done in the (mem_65MB_tb) the transcript output shows the all tests for the memory unit were successfully passed.

```
#
#         Total number of tests          27
#         Total number of pass           27
#
```

VI. Control Unit (State Machine) Design and Testing

A. *State Machine Design*

The Control Units state machine is designed to progressively increment between states (FETCH, DECODE, EXECUTE, MEMORY, and WRITE BACK) every clock cycle as long as the system is on. During the initialization and reset of the state machine the current state (state_reg which is assigned to the STATE output) is set to 2'bxx and the next state (next_state) is set to FETCH. Reset is designed to only occur when RST is set to 1'b0 and on the negative edge of the RST.

```
assign STATE = state_reg;

// initiation state
initial
begin
   state_reg = 2'bxx;
   next_state = `PROC_FETCH;
end

// reset signal
always@(negedge RST)
begin
   state_reg = 2'bxx;
   next_state = `PROC_FETCH;
end
```

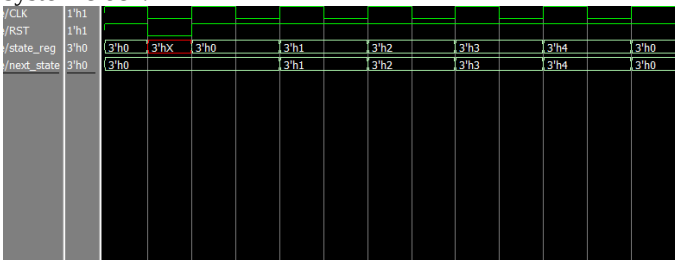
After initialization and reset, the machine is designed to switch states at every positive edge of the system clock cycle. This process done by setting the current state (state_reg) to the next_state and setting next_reg to the appropriate state determined by the state machine's state switching properties. This keeps occurring as long as the state machine is active.

```
//state switching
always@(posedge CLK)
begin

 case(STATE)
    `PROC_FETCH : next_state = `PROC_DECODE;

    `PROC_DECODE : next_state = `PROC_EXE;

    `PROC_EXE : next_state = `PROC_MEM;

    `PROC_MEM : next_state = `PROC_WB;

    `PROC_WB : next_state = `PROC_FETCH;
  endcase

  state_reg = next_state;

end
```

B. State Machine Testing

Testing of the state machine is simply making sure that the machine changes state at every positive clock edge for the duration of the simulation, as well as making sure that the initialization and reset work correctly.

Using the wave table it is easy to see that the initialization and reset operations are working as they should. As shown in the wave table, reset operations occur at the negative edge of the reset and states change at every positive edge of the system clock.



VII. R-Type Instructions

A. R-Type Instruction Design

R-Type Instructions were the simplest to implement into the control unit. Since the opcode was h'00 for all R-Type instructions, it only took one case to test if the given instruction was R-Type during the execute step. After knowing that the instruction was R-Type through the case testing, only three if statements were necessary to be able to differentiate between a jump register function, ALU function using the two register outputs R1 and R2, and ALU functions using R1 and shift amount (shamt). For all the ALU functions, the control unit did not need to differentiate which ALU function to do, since the function code (funct) in the instruction bits lets the ALU decide what to do.

```
// R-Type
6'h00 :
begin
   if(funct === 6'h08)
   begin
    PC_REG = RF_DATA_R1;
   end

   else if(funct === 6'h01 || funct === 6'h02)
   begin
    ALU_OPRN_RET = funct;
    ALU_OP1_RET = RF_DATA_R1;
    ALU_OP2_RET = shamt;
   end

   else
   begin
     ALU_OPRN_RET = funct;
     ALU_OP1_RET = RF_DATA_R1;
     ALU_OP2_RET = RF_DATA_R2;
   end
end
```

The R-Type instructions are not dealt with again until the write back stage of the control unit. At this stage, the control unit again figures out if it is an R-Type instruction through case testing and then differentiates if it is a Jump Register instruction or the other R-Type arithmetic / logical instructions. The reason this must be done is because Jump Register affects the PC and the ALU based instructions are stored in the R[rd] register.

```
//R-Type Register WriteBack
6'h00 :
begin
   if(funct === 6'h08)
     PC_REG = RF_DATA_R1;
   else
   begin
     RF_ADDR_W_RET = rd;
     RF_DATA_W_RET = ALU_RESULT;
     RF_WRITE_RET = 1'b1;
   end
end
```

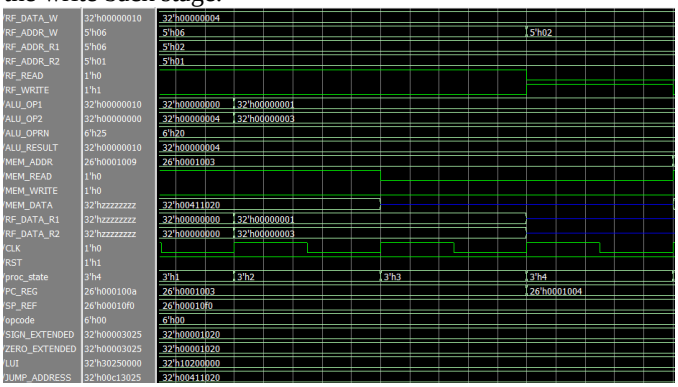
B. R-Type Instruction Testing

A self-made memory dat file was created and loaded into the Da Vinci machine to test R-Type Instructions. It consists

of several instructions such as add, sub, mul, srl, sll, and or. A jump was placed at the end to keep the machine looping this instruction set as long as it was running.

```
@0001000
20420001          //addi r[2], r[2], 0x0001;
20210003          //addi r[1], r[1], 0x0003;
20c60004          //addi r[6], r[6], 0x0004;
00411020 //loop: //add r[2], r[2], r[1];
00411022          //sub r[2], r[2], r[1];
00460820          //add r[1], r[2], r[6];
0046082c          //mul r[1], r[2], r[6];
00400882          //srl r[1], r[2], 2;
00c03081          //sll r[6], r[6], 2;
00c13025          //or  r[6], r[6], r[1];
08001003          //jmp  loop;
```

When running the test program, the wave table made it easy to see if the control unit was responding correctly to the given instruction set. Using this wave table, I could track which registers were being read to give information for the given instruction and then where the results were going during the write back stage.



VIII. I-Type Instructions

A. *I-Type Instruction Design*

I-Type Instructions were implemented by providing different cases during the execution stage of the control unit to determine which instruction to execute depending on the given opcode. All I-Type instructions besides beq and bne are implemented in the execution state of the control unit.

```
//I-Type
6'h08 :
begin
  ALU_OPRN_RET = `ALU_OPRN_WIDTH'h20;
  ALU_OP1_RET = RF_DATA_R1;
  ALU_OP2_RET = SIGN_EXTENDED;
end
6'h1d :
begin
  ALU_OPRN_RET =`ALU_OPRN_WIDTH'h2c;
  ALU_OP1_RET = RF_DATA_R1;
  ALU_OP2_RET = SIGN_EXTENDED;
end
6'h0c :
begin
  ALU_OPRN_RET = `ALU_OPRN_WIDTH'h24;
  ALU_OP1_RET = RF_DATA_R1;
  ALU_OP2_RET = ZERO_EXTENDED;
end
6'h0d :
begin
  ALU_OPRN_RET = `ALU_OPRN_WIDTH'h25;
  ALU_OP1_RET = RF_DATA_R1;
  ALU_OP2_RET = ZERO_EXTENDED;
end
6'h0a :
```

Since these instructions make use of an immediate, ZeroExtended, SignExtended, LUI, and BranchAddress needed to be created using the immediate given in the instruction code.

```
//Immediate sign extension
SIGN_EXTENDED = {{16{immediate[15]}},immediate};
//Immediate zero extension
ZERO_EXTENDED = {16'h0000, immediate};
//LUI value
LUI = {immediate, 16'h0000};
//Store 32-bit jumpaddress
JUMP_ADDRESS = {6'b0, address};
```

After being executed, I-Type instructions that control memory read and write are implemented in the memory state of the control unit. There were only two instructions that needed to be implemented in this state (lw and sw). The right instruction is determined by a case test using the given opcode.

```
case (opcode)
//LW
6'h23 :
begin
  MEM_ADDR_RET = ALU_RESULT;
  MEM_READ_RET = 1'b1;
end
//SW
6'h2b :
begin
  MEM_ADDR_RET = ALU_RESULT;
  MEM_DATA_RET = RF_DATA_R2;
  MEM_WRITE_RET = 1'b1;
end
```

Finally any I-Type instructions that change register or PC data are executed in the write back state of the control unit. The right instruction is determined by a case test using the opcode

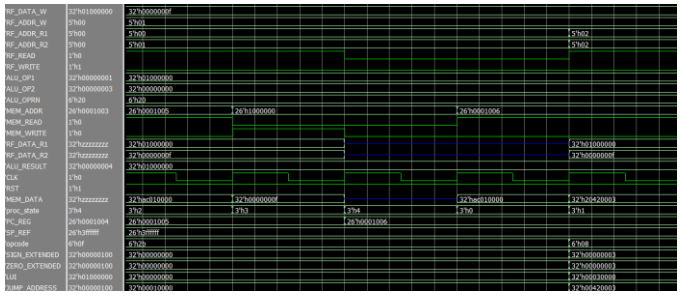
```
6'h0a :
begin
 RF_ADDR_W_RET = rt;
 RF_DATA_W_RET = ALU_RESULT;
 RF_WRITE_RET = 1'b1;
end
6'h04 :
begin
 if(RF_DATA_R1 == RF_DATA_R2)
   PC_REG = PC_REG + SIGN_EXTENDED;
end
```

B. *I-Type Instruction Testing*

A simple instruction set was created to test out the I-Type Instructions and make sure that the control unit did the appropriate executions depending on the given instruction set. Arithmetic, logic, and branching instructions were tested. This simple instruction set was loaded into the memory of the Da Vinci system.

```
@0001000
20420001 //          addi r[2], r[2], 0x0001;
20210003 //          addi r[1], r[1], 0x0003;
20c60004 //          addi r[6], r[6], 0x0004;
3C000100 // loop:   lui  r[0], 0x0100;
74210005 //          muli r[1], r[1], 0x0005;
AC010000 //          sw   r[0], r[1], 0x0000;
20420003 //          addi r[2], r[2], 0x0003;
10460001 //          beq  r[2], r[6], 0x0001;
3442000a //          ori  r[2], r[2], 0x000a;
08001003 //          jmp  loop;
```

After loading the instructions into the system, the wave table was used to analyze how the control unit responded. By carefully following the wave table it can be seen how the control unit follows the instructions given in the test dat file (Itype_test.dat) accessing memory and registers at the correct addresses at the correct times.



IX. J-Type Instructions

A. *J-Type Instruction Design*

J-Type operations are fairly simple to implement since there are only four operations in total. Only the push operation (opcode: 1b) takes part in the execution state of the control unit. The control unit is programmed to set the address of DATA_R1 to register 0. The control unit knows when this state occurs by using case testing from the opcode.

```
//J-Type
6'h1b :
begin
RF_ADDR_R1_RET = 0;
end
```

Depending on the specified J-Type instruction actions can be taken in the mem state and / or write back state of the control unit. Behavior was written for both push and pop instructions in the mem state of the control unit, since push writes to memory and pop reads from memory. Which function that needs to be executed is determined by a test case using the opcode.

```
//PUSH
6'h1b :
begin
  MEM_ADDR_RET = SP_REF;
  MEM_DATA_RET = RF_DATA_R1;
  MEM_WRITE_RET = 1'b1;
  SP_REF = SP_REF - 1;
end
//POP
6'h1c :
begin
  SP_REF = SP_REF + 1;
  MEM_ADDR_RET = SP_REF;
  MEM_READ_RET = 1'b1;
end
```

All J-Type instructions except for push are implemented in the write back state since they all have steps that write to the registers or change the program counter (PC_REG).

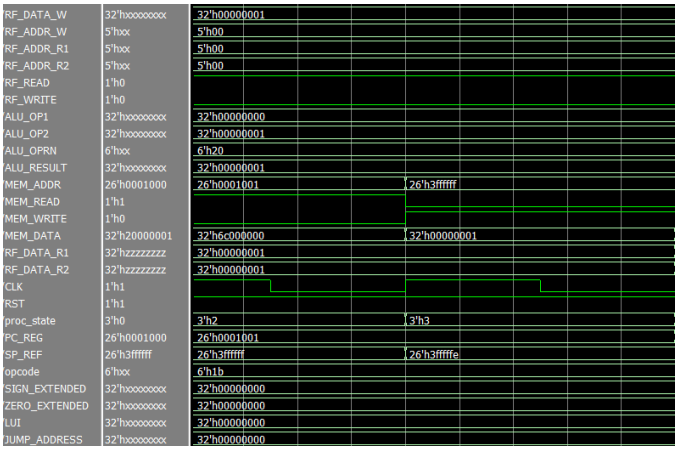
```
//J-Type
6'h02 : PC_REG = JUMP_ADDRESS;

6'h03 :
begin
  RF_ADDR_W_RET = 31;
  RF_DATA_W_RET = PC_REG;
  RF_WRITE_RET = 1'b1;
  PC_REG = JUMP_ADDRESS;
end
6'h1c :
begin
  RF_ADDR_W_RET = 0;
  RF_DATA_W_RET = MEM_DATA;
  RF_WRITE_RET = 1'b1;
end
```

B. *J-Type Instruction Testing*

To test the J-Type instructions a simple instruction set was made and loaded into the Da Vinci's memory module. The instruction set consisted of testing jump and link (jal), push, and pop. It was not really necessary to test jump (jmp) as it had already been used in previous testing.

```
@0001000
20000001 //          addi r[0], r[0], 0x0001;
6c000000 //          push;
20000007 //          addi r[0], r[0], 0x0007;
70000000 //          pop;
0c001000 //          jal 0x001002;
```

When executing the code and analyzing the wave table generated by Da Vinci running, it was easy to see that the control unit was executing the proper instructions according to the data given. Push and pop were properly affecting the stack pointer (SP_REF) and jal was properly affecting the program counter (PC_REG) and uploading the previous PC info to register 31 (R[31]).



X. CONCLUSION

In conclusion, programming a behavioral model of the Da Vinci v1.0 in Verilog offers a great view into how a basic processor running the ‘CS147DV’ instruction set works in conjunction with a memory module. Building the control unit, register, and ALU separately then wiring them and having them interact under a single processor offers a lot of insight to their specific duties when executing instructions.